04. Cloud Migration

0. 사전 준비

Local 환경에서 두 개의 Redis instance가 필요합니다.
하나는 6379, 하나는 7777 port 입니다.
Docker를 사용한다면 아래와 같이 간단히 테스트 용으로 생성합니다.

pom.xml

```
docker run -d -p 6379:6379 redis

Redis-Cli
docker run -it --link ${containerNames}:redis --rm redis redis-cli -h redis -p 6379

docker run -d -p 7777:7777 redis --port 7777
Redis-Cli  docker run -it --link ${containerNames}:redis --rm redis redis-cli -h redis -p 7777
```

1. Spring Session 설정

클라우드 환경에서 세션 클러스터링을 사용하기 위해 Redis를 기반으로 하는 Spring Session 적용

Maven 의존성 추가

pom.xml

```
<dependency>
            <groupId>biz.paluch.redis</groupId>
            <artifactId>lettuce</artifactId>
            <version>3.5.0.Final</version>
        </dependency>
 <dependency>
      <groupId>org.springframework.session</groupId>
      <artifactId>spring-session-data-redis</artifactId>
      <version>1.3.5.RELEASE</version>
  </dependency>
```

→ Lettuce는 Netty를 기반으로 하는 Redis client 입니다.
기존에 많이 사용하는 Jedis는 Blocking IO를 기반으로 하는 반면
Lettuce는 Non-Blocking IO를 기반으로 하여 TPS/CPU/Connection 등 지표에서 Jedis 보다 월등히 좋은 성능을 보입니다

또한 Jedis보다 활발하게 Github Repository에 커밋이 쌓이고 있어 향후에도 더욱 발전 가능성이 큰 라이브러리 입니다.

스프링 설정

application-context.xml

```
<bean id="lettuceConnectionFactory" class="org.springframework.data.redis.connection.lettuce.
LettuceConnectionFactory"
          p:hostName="localhost" p:port="6379" p:database="0" p:password=""/>

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="lettuceConnectionFactory" />
</bean>

<bean class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttpSessionConfiguration"/>
```

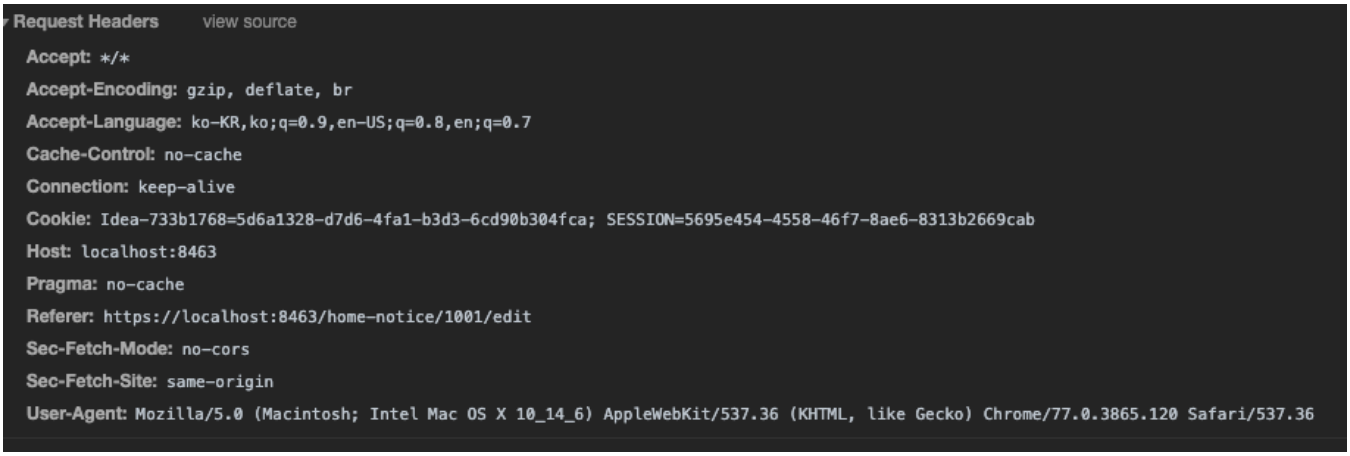
web.xml 설정 (Character encoding 필터보다 반드시 뒤에 위치해야합니다)

web.xml

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Spring session은 별다른 설정이 없다면 CookieHttpSessionStrategy를 사용합니다. 아래와 같이 Request Header의 cookie에 Session cookie가 붙어있습니다.



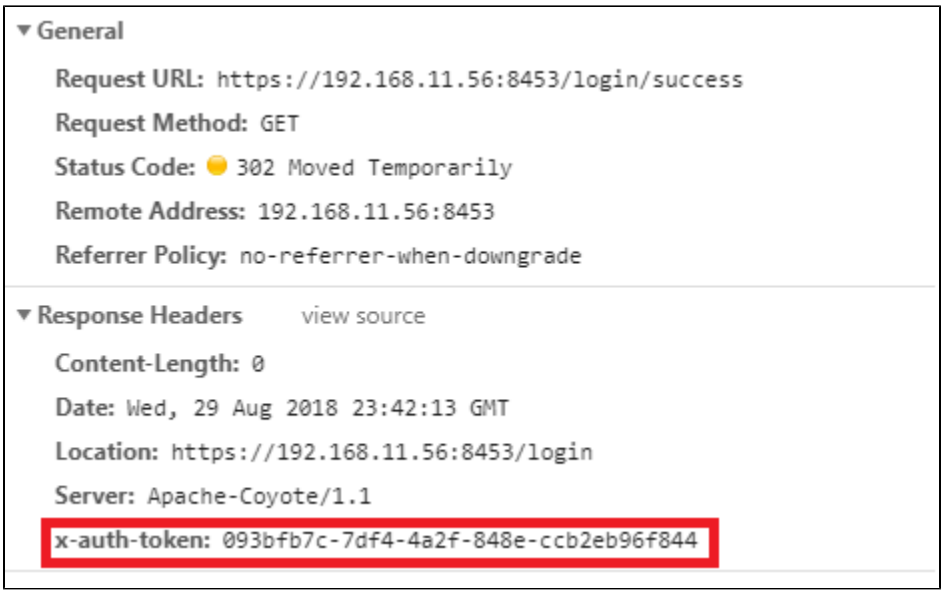
만약 Cookie를 사용하는 것이 싫다면 HeaderHttpSessionStrategy를 RedisHttpSessionConfiguration에 주입하면 됩니다.

HeaderHttpSessionStrategy

```
<bean class="org.springframework.session.web.http.HeaderHttpSessionStrategy" name="sessionStrategy" />

<bean class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttpSessionConfiguration" >
   <property name="httpSessionStrategy" ref="sessionStrategy"/>
</bean>
```

만약 위의 설정을 토대로 진행할 경우 Login 요청 후 Response Header에 담긴 x-auth-token을 받아와 AccessToken처럼 매번 Request Header에 실어 보내주는 로직을 추가적으로 개발해야 합니다.



추가적인 로직 개발에 비용이 클 것으로 예상하여 CTMS는 CookieHttpSessionstrategy를 사용합니다.

2. Spring Security ACL cache의 CacheManager 구현체를 RedisCacheManager로 전환

Session clustering과 마찬가지로 ACL cache도 clustering 설정이 되어야 합니다.

따라서 아래의 작업을 추가로 진행합니다.

Maven 의존성 추가

pom.xml

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>1.7.0.RELEASE</version>
</dependency>
```

스프링 설정

application-context.xml

```
<bean id="cacheLettuceConnectionFactory" class="org.springframework.data.redis.connection.lettuce.
LettuceConnectionFactory"
      p:hostName="localhost" p:port="7777" p:database="0" p:password=""/>

<bean id="cacheRedisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="cacheLettuceConnectionFactory" />
</bean>
```

ACL cacheManager 설정 변경

acl-cache.xml

```
<cache:annotation-driven/>

<jee:jndi-lookup id="nativeCacheManager"
                 expected-type="org.infinispan.manager.EmbeddedCacheManager"
                 jndi-name="java:#{stageProperties['jndi.nativeCacheManager']}"/>

<bean id="cacheManager" class="kr.co.crscube.ctms.security.acl.AsyncFlushableCacheManager">
    <constructor-arg ref="cacheRedisTemplate" />
</bean>
```

여기서 주의할 점은 현재 LettuceConnectionFactory 타입의 Bean이 두개가 선언되어 있어 아래와 같은 상황이 발생할 수 있습니다.

JBoss를 띄워보면 아래와 같은 메시지를 띄우고 뻗어버린다.

> *14:50:49,127 JBWEB000287: Exception sending context initialized event to listener instance of class org.springframework.web.context.ContextLoaderListener*
> *14:50:49,164 JBWEB001103: Error detected during context start, will stop it*
> *14:50:49,182 Closing Spring root WebApplicationContext*
> *14:50:49,186 MSC000001: Failed to start service jboss.web.deployment.default-host./*
> *...*
> *[2019-10-24 02:50:49,290] Artifact ctms:war exploded: Error during artifact deployment. See server log for details.*

정확한 판단을 위해 server.log를 확인하였고 아래와 같은 로그를 확인할 수 있었다.

> *Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type [org.springframework.data.redis.connection.RedisConnectionFactory] is defined: expected single matching bean but found 2: lettuceConnectionFactory,cacheLettuceConnectionFactory at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:)*

로그 메시지를 파악하면 두 개의 RedisConnectionFactory 타입의 Bean이 감지되어 Qualifying이 불가한 것인데요 (LettuceConnectionFactory는 RedisConnectionFactory의 구현체입니다)

Spring Session 설정의 RedisHttpSessionConfiguration Bean 안에서 생성되는 redisMessageListenerContainer Bean의 생성자 0번째에서 아래와 같이 RedisConnectionFactory에 대한 의존성을 주입받고 있습니다.

RedisHttpSessionConfiguration

```
@Bean
public RedisMessageListenerContainer redisMessageListenerContainer(
    RedisConnectionFactory connectionFactory,
    RedisOperationsSessionRepository messageListener) {

  RedisMessageListenerContainer container = new RedisMessageListenerContainer();
  container.setConnectionFactory(connectionFactory);
  if (this.redisTaskExecutor != null) {
    container.setTaskExecutor(this.redisTaskExecutor);
  }
  if (this.redisSubscriptionExecutor != null) {
    container.setSubscriptionExecutor(this.redisSubscriptionExecutor);
  }
  container.addMessageListener(messageListener,
      Arrays.asList(new PatternTopic("__keyevent@*:del"),
          new PatternTopic("__keyevent@*:expired")));
  container.addMessageListener(messageListener, Arrays.asList(new PatternTopic(
      messageListener.getSessionCreatedChannelPrefix() + "*")));
  return container;
}
```

따라서 Spring Session 전용 LettuceConnectionFactory의 Bean name을 connectionFactory로 변경해주어 RedisMessageListenerContainer Bean에게 주입할 의존성을 Qualifying 해줍니다.

acl-cache.xml

```
<context:annotation-config />

<!-- RedisHttpSessionConfiguration connectionFactory    -->
<!-- Spring session  Redis config -->
<bean id="connectionFactory" class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory"
      p:hostName="localhost" p:port="6379" p:database="0" p:password=""/>

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>
<bean class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttpSessionConfiguration"/>

<!-- Spring Security ACL  Redis config -->
<bean id="cacheLettuceConnectionFactory" class="org.springframework.data.redis.connection.lettuce.
LettuceConnectionFactory"
      p:hostName="localhost" p:port="7777" p:database="0" p:password=""/>

<bean id="cacheRedisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="cacheLettuceConnectionFactory" />
</bean>
```

3. File Upload Issue

Spring Session을 적용하다보면 파일업로드가 정상적으로 처리되지 않는 문제가 발생 할 수 있습니다.

이전에 등록한 springSessionRepositoryFilter에서 CookieHttpSessionStrategy를 사용하므로
Request를 미리 파싱해버리기 때문에 DispatcherServlet에서 MultipartReolver가 Resolve 할 Multipart들이 소실되기 때문입니다.
(https://github.com/spring-projects/spring-session/issues/649)

따라서 springSessionRepositoryFilter의 앞에서 먼저 Multipart 데이터들을 처리할 수 있어야 하는데요 MultipartFilter가 이부분에 적합합니다.

MultipartFilter 적용

아래와 같이 springSessionRepositoryFilter의 선언 앞에 MultipartFilter를 먼저 등록합니다.

web.xml (MultipartFilter)

```
<filter>
        <filter-name>characterSetEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>characterSetEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>multipartFilter</filter-name>
        <filter-class>org.springframework.web.multipart.support.MultipartFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>multipartFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>springSessionRepositoryFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSessionRepositoryFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- Spring Security Chain filter -->
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

MultipartFilter의 구현체를 보면 lookupMultipartResolver 메소드에서 WebApplicationContextf로 부터 MultipartResolver를 lookup 합니다.
getMultipartResolverBeanName는 multipartResolverBeanName field에 대한 getter입니다.

lookupMultipartResolver

```
protected MultipartResolver lookupMultipartResolver() {
		WebApplicationContext wac = WebApplicationContextUtils.getWebApplicationContext
(getServletContext());
		String beanName = getMultipartResolverBeanName();
		if (wac != null && wac.containsBean(beanName)) {
			if (logger.isDebugEnabled()) {
				logger.debug("Using MultipartResolver '" + beanName + "' for MultipartFilter");
			}
			return wac.getBean(beanName, MultipartResolver.class);
		}
		else {
			return this.defaultMultipartResolver;
		}
	}
```

여기서 multipartResolverBeanName에 대해 별도로 설정을 하지 않았으므로 DEFAULT_MULTIPART_RESOLVER_BEAN_NAME이 적용됩니다.
DEFAULT_MULTIPART_RESOLVER_BEAN_NAME은 아래처럼 **filterMultipartResolver**입니다.

```
 *
 * @author Juergen Hoeller
 * @since 08.10.2003
 * @see #setMultipartResolverBeanName
 * @see #lookupMultipartResolver
 * @see org.springframework.web.multipart.MultipartResolver
 * @see org.springframework.web.servlet.DispatcherServlet
 */
public class MultipartFilter extends OncePerRequestFilter {

	public static final String DEFAULT_MULTIPART_RESOLVER_BEAN_NAME = "filterMultipartResolver";

	private final MultipartResolver defaultMultipartResolver = new StandardServletMultipartResolver();

	private String multipartResolverBeanName = DEFAULT_MULTIPART_RESOLVER_BEAN_NAME;
```

따라서 아래와 같이 **ServletContext가 아닌 RootContext(ApplicationContext)**에 **filterMultipartResolver**라는 이름으로 Bean을 등록합니다.
그리고 Filter에서 이미 Multipart에 대해 Resolve를 진행 했으므로 DispatcherServlet에서의 작업은 필요 없으니 **기존에 설정 된 ServletContext 내에 MultipartResolver 설정은 제거합니다**.

ApplicationConteext

```
    <!--
        MultipartFilter   Root Application Context   Multipart Resolver
         MultipartFilter       Servlet Context   Multipart Resolver
     -->
    <bean id="filterMultipartResolver" class="org.springframework.web.multipart.commons.
CommonsMultipartResolver">
        <property name="maxUploadSize" value="102400000"/>
        <property name="defaultEncoding" value="UTF-8"/>
    </bean>
```

ServletContext

```
 <!-- MultipartFilter   Root Application Context Multipart Resolver  -->
    <!--<bean id="multipartResolver" class="org.springframework.web.multipart.support.
StandardServletMultipartResolver"/>-->
    <!--<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"
>-->
        <!--<property name="maxUploadSize" value="102400000"/>-->
        <!--<property name="defaultEncoding" value="UTF-8"/>-->
    <!--</bean>-->
```

위 설정을 진행하지 않으면 파일 업로드와 함께 특정 데이터들을 전송할 때 Encoding이 깨질 수 있습니다.

MultipartFilter는 MultipartResolver를 lookup 할 때 filterMultipartResolver라는 이름을 찾지 못하면 StandardMultipartResolver를 사용하는데요
해당 구현체는 Multipart와 함께 넘어온 파라미터들에 대해서 인코딩 작업을 하지 않기때문입니다.

CTMS에선 File과 함께 meta data를 전송하는 부분이 있어 반드시 CommonsMultipartResolver를 사용해야 합니다.